

### IN THE SPECIFICATION

The paragraph beginning at page 2, line 3, is amended as follows:

A high level language loop specifies a computation to be performed iteratively on different elements of some organized data structures, such as arrays, structures, records and so on. Computations in each iteration typically translate into loads (to access the data), computations (to compute on the data loaded), and stores (to update the data structures in memory). Achieving higher performance often entails performing these actions, related to different iterations, concurrently. To do so, loads from many successive iterations often have to be performed before stores from current iterations. When the data structures being accessed are done so indirectly (either through pointer or via indirectly obtained indices) the dependence between stores and loads is dependent on data values of pointers or indices produced at run time. Therefore, at compile time there exists a "probable" dependence. Probable store-to-load dependence between iterations in a loop creates the ambiguity that prevents the compiler from hoisting the next iteration's loads and the dependent computations above the stores from the prior iteration(s). The compiler cannot assume the absence of such dependence, since ignoring such a probable dependence (and hoisting the load) will lead to compiled code that produces incorrect results.

The paragraph beginning on page 2, line 29, is amended as follows:

A number of compilation techniques have been developed to improve the efficiency of loop computations by increasing instruction-level parallelism (ILP). One such method is software-pipelining, which improves the performance of a loop by overlapping the execution of several independent iterations. The number of cycles between the start of successive iterations in software-pipelining is called the initiation interval ~~initiation interval~~, which is the greater of the resource initiation interval ~~resource initiation interval~~ and the recurrence initiation interval ~~recurrence initiation interval~~. The resource initiation interval is based on the resource usage of the loop and the available processor resources. The recurrence initiation interval of the loop is based on the number of cycles in the dependence graph for the loop and the latencies of a

processor. A higher ~~Maximum~~ instruction-level parallelism for the loop can be ~~[[is]]~~ realized if the recurrence initiation interval of the loop is less than or equal to its resource initiation interval. Typically, this condition is not satisfied for loops whose computations involve sparse arrays/matrices. The body of such a loop typically starts with a load whose address in itself is an element of another array (called the index array) and ends with a store whose address is an element of the index array. In the absence of static information, the compiler does not know about the contents of the elements of the index array. Hence, it must assume that there is a loop-carried dependence edge from the store in one iteration to the load in the next iteration. This makes the recurrence initiation interval much higher than resource initiation interval.

The paragraph beginning at page 5, line 17, is amended as follows:

The following example illustrates the problem of recurrence initiation interval being higher than resource initiation interval in a vectorizable loop requiring computation of sparse arrays/matrices:

First consider a loop performing computations on arrays that are directly accessed:

The paragraph beginning at page 5, line 16, is amended as follows:

Now consider the The following loop, which represents sparse "Gather Vector Add," illustrates a common operation in sparse matrix computations:

The paragraph beginning at page 6, line 10, is amended as follows:

Following is an example Intel® Itanium® architecture ~~the~~ code generated for the above loop, without software-pipelining:

The paragraph beginning at page 6, line 12, is amended as follows:

```
.b1_1:                                // 21 cycles per iteration
{ .mmi
    ld8    r9=[r32],8                // cycle 0:  load b[i]
    ldfs   f8=[r34],4                // cycle 0:  load c[i]
    nop.i  0 ;;
} { .mmi
    shladd r2=r9,2,r33 ;;            // cycle 2:  address of (a[b[i]])
    ldfs   f7=[r2]                  // cycle 3:  load a[b[i]]
    nop.i  0 ;;
```

---

```

} { .mfi
    nop.m 0
    fma.s f6=f7,f1,f8      // cycle 12: add a[b[i]] + c[i]
    nop.i 0 ;;
} { .mib
    stfs  [r2]=f6          // cycle 20: store add result a[b[i]] =a[b[i]]
+c[i]
    nop.i 0
    br.cloop.sptk .b1_1 ;; // cycle 20:

```

The paragraph beginning at page 6, line 34, is amended as follows:

One possible solution for the compiler to reduce recurrence initiation interval is to use the technique of data speculation, in processor architectures such as the Intel® Itanium® architecture, which provides a mechanism for the compiler to break, possible but unlikely, store-to-load memory dependencies. It is possible to efficiently software-pipeline the above example loop using data speculation. Such a loop will have a scheduled initiation interval of 3 cycles, since a `chk.a` instruction will be introduced into the loop to recover from a speculation failure (i.e., when the “possible” dependence is a “true” dependence). The performance of the loops with and without data speculation when using a processor such as the Intel® Itanium® processor is typically much better with data speculation than without it, for example as much as 1/5 as many ~~few~~ cycles for the former as compared ~~opposed~~ to the latter.

The paragraph beginning at page 7, line 5, is amended as follows:

However, it is generally not known how often the index array elements collide, i.e.,  $b[i] == b[j]$  for  $i \neq j$  (i.e., the possible dependence becomes a true dependence). If the `chk.a` ~~operates~~ fires at least 15 times in this loop (either because of true collisions in the index array  $b[i]$ , or even because of false Advanced Load Address Table (ALAT) conflicts, which is a conflict detection structure in the Intel® Itanium® processor), the performance gain due to data speculation will disappear and the resultant performance will be actually be worse than that of the software-pipelined loop without data speculation. This is because, for example, the penalty of a `chk.a` instruction miss is about 100 cycles in the Intel® Itanium® processor ~~processors~~.

The paragraph beginning at page 7, line 14, is amended as follows:

3. The worst-case scenario can be that  $b[M] == b[M-1]$ . In such a case, the just computed ~~loaded~~ value of  $a[b[M-1]]$  is used to compute the new value of

a [b [M] ]. Hence the recurrence initiation interval of this loop is 5 cycles, since the latency of one Floating Point Multiply Add (FMA) instruction to another FMA is 5 cycles for a processor such as a processor having the Intel® Itanium® architecture.

The paragraph beginning at page 8, line 20, is amended as follows:

At 110, sparse array matrix code is transformed to perform a run time dependency check using a predetermined number of prior computed values. The following example illustrates transformed code for using the source code illustrated earlier:

The paragraph beginning at page 8, line 23, is amended as follows:

```
b0 = -1;  // Initialize to an illegal value for b[i]
b1 = -1;  // ditto
b2 = -1;  // ditto

for (i = 0; i < 100; i++) {

    b3 = b[i];
    c3 = c[i];

    if (b3 == b2) {
        a3 = a2 + c3;
    } else if (b3 == b1) {
        a3 = a1 + c3;
    } else if (b3 == b0) {
        a3 = a0 + c3;
    } else {
        a3 = a[b3] + c3;
    }
    a[b[i]] = a3;

    a0 = a1;  a1 = a2;  a2 = a3;  // mcopy instructions
```

~~instructions~~~~b0 = b1;  b1 = b2;  b2 = b3;  // mcopy instructions~~~~instructions~~~~}~~

The paragraph beginning at page 9, line 30, is amended as follows:

Software-pipelining the above example transformed code results in a software-pipelined loop with an initiation interval of 5 cycles. As mentioned above, software-pipelining with data speculation for this loop will give us an initiation interval of 3 cycles. However, this is under ideal conditions, when there are no real or false ALAT conflicts. In this loop with 100 iterations, even when there are just 2 ALAT conflicts (again assuming the penalty of a `chk.a` miss to be 100 cycles), the real real initiation interval of the software-pipelining loop with data speculation will be 5 cycles. Then the performance of the loop with the novel method will equal that of the loop with software-pipelining data speculation. When there are 3 or more ALAT conflicts, the real initiation interval of the software-pipelining loop with data speculation will be 6 cycles or more, and this example embodiment can result in improved performance.

The paragraph beginning at page 10, line 5, is amended as follows:

In the absence of additional information, the software-pipelining embodiments of the invention ~~deliver a performance substantially better than that obtained via conservative scheduling and only slightly better than that obtained via scheduling using data speculation. It~~ may perform better than data speculation, even if the probability of the loop-carried store-to-load dependence is very small.

The paragraph beginning at page 10, line 10, is amended as follows:

In some embodiments, resource initiation interval is based on resource usage of the dependence loop and available processor resources. Also in these embodiments, recurrence initiation interval is based on the number of cycles in the dependence graph for the loop instruction in the dependence loop body and latencies of the processor.

The paragraph beginning at page 11, line 10, is amended as follows:

Referring now to FIG. 2, there is illustrated an example embodiment 200 of software-pipelining transformed sparse array matrix code illustrated at 110 in FIG.1. At 210, this example embodiment 200 computes ~~forms~~ a predetermined number of variables based on a virtual unrolling factor. In some embodiments, the predetermined number of variables is computed using the equation:

The paragraph beginning at page 11, line 15, is amended as follows:

$$M = \text{Ceiling} \left( \frac{\text{Recurrence II/FMA Latency}}{\text{Recurrence II/FMA Latency}} \right) - 1$$

where ~~Where~~  $M$  is the virtual unrolling factor, i.e., the number of previous iterations from which the computed values have been temporarily held in registers.

The paragraph beginning at page 11, line 22, is amended as follows:

At 220, the computed ~~formed~~ predetermined number of variables is initialized. In the example previously given above, the ~~predetermined number of variables is initialized by creating~~ variables  $b_0, b_1, \dots, b_{M-1}$  and  $a_0, a_1, \dots, a_{M-1}$  are computed and ~~initializing~~  $b_0, b_1, \dots, b_{M-1}$  are initialized to an illegal value for the  $b$  array, say to  $-1$ . In these embodiments, initialization is done outside the loop.

The paragraph beginning at page 12, line 1, is amended as follows:

At 240, again inside the loop body, the prior computed values are assigned to a predetermined number of registers. In the running example embodiment, a nested, if-then-else statement is created inside the loop. If  $(b_{M-1} == b_M)$ , then the variable  $a_M$  is assigned the value of the right hand side (RHS) computed using the value of  $a_{M-1}$  rather than the value of  $a[b_M]$ . Similarly, the nested if statement continues by comparing  $(b_{M-2} == b_M)$ . If this is true, then the variable  $a_M$  is assigned the value of the RHS computed using the value of  $a_{M-2}$  rather than the value of  $a[b_M]$ . This is performed until the comparison  $(b_0 == b_M)$  is complete. When none of the above comparisons are true (the final else clause in the nested if-then-else statement), that

means the current index array element is not equal to any of the previous  $(M - 1)$  elements. In this case, the variable  $a_M$  is assigned the value of the RHS computed using the value of  $a[b_M]$ .

The paragraph beginning at page 12, line 18, is amended as follows:

As described above, multiple iterations are software-pipelined to reduce recurrence initiation interval, perhaps by overlapping execution of dependence loop instructions in multiple dependent iterations in the sparse arrays/matrices. In these embodiments, software-pipelining includes parallelizing the multiple iterations using the prior computed values to reduce recurrence initiation interval based on the run time dependency check. In these embodiments, dependence loop body instructions are parallelized between the subsequent independent iterations. Loop body instructions are based on clock cycles. The above-described run time dependency check and parallelizing the dependence loop instruction improves the efficiency of loop computations by increasing the instruction-level parallelism.

The paragraph beginning at page 14, line 1, is amended as follows:

The loop-carried memory dependence from the store of  $a[b[i]]$  to the load of  $a[b[i+1]]$  in the next iteration no longer exists. This is because the stored values of  $a[b[i-3]]$ ,  $a[b[i-2]]$ , and  $a[b[i-1]]$  are remembered. Because of this transformation, only the loop-carried memory dependence from the store of  $a[b[i]]$  460 in the first column 410 to the load of  $a[b[i+4]]$  470 in the fifth column 450 exists, i.e. the loop-carried memory dependence is now across 4 iterations. FIG. 4 shows the need for 5 iterations, i.e., between the first iteration and the fifth iteration the dependents are taken care of in the second through fourth iterations. The ~~value for the load of  $a[b[i+4]]$~~   $a[b[i+4]]$  in the fifth iteration is obtained from the store of  $a[b[i]]$  in the first iteration. That is why five iterations are needed in the software-pipelining of this example embodiment of the present invention.

The paragraph beginning at page 14, line 12, is amended as follows:

This technique reduces recurrence initiation interval of sparse arrays/matrices by virtual unrolling. i.e., keeps the values computed in previous iterations in rotating registers and uses them if needed as described above. Some embodiments of the invention exploit ~~The invention~~

**PRELIMINARY AMENDMENT**

Serial Number: 10/612724

Filing Date: June 30, 2003

Title: SYSTEM AND METHOD FOR SOFTWARE-PIPELINING OF LOOPS WITH SPARSE MATRIX ROUTINES

Assignee: Intel Corporation

Page 10

Dkt: 884.889US1 (INTEL)

---

exploits features, such as predication, rotating registers, and special branches for software-pipelining, provided in a processor such as, but not limited to, those of the Intel® Itanium® processor family architecture.